Sort It Out with Monotonicity

Translating between Many-Sorted and Unsorted First-Order Logic

Koen Claessen, Ann Lillieström, Nicholas Smallbone

Chalmers University of Technology, Gothenburg, Sweden
{koen,annl,nicsma}@chalmers.se

Abstract. We present a novel analysis for sorted logic, which determines if a given sort is *monotone*. The domain of a monotone sort can always be extended with an extra element. We use this analysis to significantly improve well-known translations between unsorted and many-sorted logic, making use of the fact that it is cheaper to translate monotone sorts than non-monotone sorts. Many interesting problems are more naturally expressed in many-sorted first-order logic than in unsorted logic, but most existing highly-efficient automated theorem provers solve problems only in unsorted logic. Conversely, some reasoning tools, for example model finders, can make good use of sort-information in a problem, but most problems today are formulated in unsorted logic. This situation motivates translations in both ways between many-sorted and unsorted problems. We present the monotonicity analysis and its implementation in our tool *Monotonox*, and also show experimental results on the TPTP benchmark library.

1 Introduction

Many problems are more naturally expressed in many-sorted first-order logic than in unsorted logic, even though their expressive power is equivalent. However, none of the major automated theorem provers for first-order logic can deal with sorts. Most problems in first-order logic are therefore expressed in unsorted logic.[1] However, some automated reasoning tools (such as model finders) could greatly benefit from sort information in problems.

This situation motivates the need for translations between sorted and unsorted first-order logic: (1) users want to express their problems in sorted logic, whereas many tools only accept unsorted logic; (2) some tool developers want to work with sorted logic, whereas the input problems are mostly expressed in unsorted logic. For example, a model finder for a sorted logic has more freedom than for an unsorted logic: it can find domains of different sizes for different sorts, and apply symmetry reduction for each sort separately.

[1] Indeed, only recently was a collection of many-sorted first-order problems added to the TPTP [11].

In this paper, we describe automated ways of translating back and forth between many-sorted and unsorted first-order logic. We use a novel *monotonicity* analysis to improve on well-known existing translations. In short, a sort is *monotone* in a problem if, for any model, the domain of that sort can be made larger without affecting satisfiability. The result of the translation for monotone sorts turns out to be much simpler than for non-monotone sorts. The monotonicity analysis and the translations are implemented in a tool called Monotonox.

To explain the problem we solve, and how monotonicity helps us, we will use the following running example.

Example 1 (monkey village). There exists a village of monkeys, with a supply of bananas. Every monkey must have at least two bananas to eat. A banana can not be shared among two monkeys. To model this situation we introduce two sorts, one of monkeys and one of bananas. We need a predicate $\mathsf{owns} \in \mathit{monkey} \times \mathit{banana} \to o$ that says which monkey owns each banana, and Skolem functions banana_1 and $\mathsf{banana}_2 \in \mathit{monkey} \to \mathit{banana}$ to witness the fact that each monkey has two bananas. We use the following four axioms:

$$\forall M \in \mathit{monkey}.\, \mathsf{owns}(M, \mathsf{banana}_1(M)) \tag{1}$$

$$\forall M \in \mathit{monkey}.\, \mathsf{owns}(M, \mathsf{banana}_2(M)) \tag{2}$$

$$\forall M \in \mathit{monkey}.\, \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M) \tag{3}$$

$$\begin{aligned} \forall M_1, M_2 \in \mathit{monkey}, B \in \mathit{banana}.\, (&\mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \\ &\Longrightarrow M_1 = M_2) \end{aligned} \tag{4}$$

We use a simple but standard many-sorted first-order logic, in which sorts α have a non-empty domain $D(\alpha)$, all symbols have exactly one sort, there are no subsorts, and equality is only allowed between terms of the same sort.

If we want to use a standard reasoning tool for unsorted logic (for example a model finder) to reason about the monkey village, we need to translate the problem into unsorted logic. Automated reasoning folklore [12] suggests three alternatives:

Sort predicates The most commonly used method is to introduce a new unary predicate P_α for every sort α that is used in the sorted formula [5]. All quantification over a sort α is translated into unsorted quantification bounded by the predicate P_α. Furthermore, for each function or constant symbol in the problem, we have to introduce an extra axiom stating the result sort of that symbol. For example, the first axiom of example 1 translates to

$$\forall M.\, (\mathsf{P}_{\mathit{monkey}}(M) \to \mathsf{owns}(M, \mathsf{banana}_1(M))$$

and we have to add axioms like $\forall M.\, \mathsf{P}_{\mathit{banana}}(\mathsf{banana}_1(M))$ for each function symbol. Moreover, to rule out the possibility of empty sorts, we sometimes need to introduce axioms of the form $\exists X.\, \mathsf{P}_\alpha(X)$.

Although conceptually simple, this translation introduces a lot of clutter which affects most theorem provers negatively: one extra predicate symbol for each sort, one axiom for each function symbol, and one extra literal for each variable in each clause.

Sort functions An alternative translation introduces a new function symbol f_α for each sort α. The translation applies f_α to any subterm of sort α in the sorted problem. The aim is to have the image of f_α in the unsorted problem be the domain of α in the sorted problem; f_α thus maps any arbitrary domain element into a member of the sort α. For example, using sort functions, the first axiom of example 1 translates to

$$\forall M.\, \mathsf{owns}(\mathsf{f}_{monkey}(M), \mathsf{f}_{banana}(\mathsf{banana}_1(\mathsf{f}_{monkey}(M))))$$

No additional axioms are needed. Thus, this translation introduces a lot less clutter than the previous translation. Still, the performance of theorem provers is affected negatively, and it depends on the theorem prover as well as the problem which translation works best in practice.

Sort erasure The translation which introduces least clutter of all simply *erases* all sort information from a sorted problem, resulting in an unsorted problem. However, while the two earlier mentioned translations preserve satisfiability of the problems, sort erasure does not, and is in fact unsound. Let us see what happens to the monkey village example. Erasing all the sorts, we get

$$\begin{gathered}
\forall M.\, \mathsf{owns}(M, \mathsf{banana}_1(M)) \\
\forall M.\, \mathsf{owns}(M, \mathsf{banana}_2(M)) \\
\forall M.\, \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M) \\
\forall M_1, M_2, B.\, (\mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 = M_2)
\end{gathered}$$

This new problem *has no finite model*, even though the sorted problem does! The reason is that, if the domain we choose has finite size k, we are forced to have k monkeys and k bananas. But a village of k monkeys must have $2k$ bananas, so this is impossible unless the domain is infinite (or empty, which we disallow). So, sort erasure does not preserve finite satisfiability, as shown by the example. In fact, it does not even preserve satisfiability.

Related work The choice seems to be between translations that are sound, but introduce clutter, and a translation that introduces no clutter but is unsound. Automated theorem provers for unsorted first-order logic have been used to reason about formulae in Isabelle [8, 9]. The tools apply sort erasure, and investigate the proof to see if it made use of unsound reasoning. If that happens they can use a sound but inefficient translation as a fall-back. A similar project using AgdaLight [1] uses sort erasure but, following [12], proposes that the theorem prover be restricted to not use certain rules (i.e. paramodulation on variables), leading to a sound (but possibly incomplete) proof procedure.

Monotonicity has been studied for higher-order logic [2] to help with pruning the search space when model finding. While the intention there is the same as ours and there are similarities between the approaches, the difference in logics changes the problem dramatically. For example, we infer that any formula without $=$ is monotone, which is not true in higher-order logic. Monotonicity is also related to the ideas of stable infiniteness and smoothness [10] in combining theories in SMT; it would be interesting to investigate this link further.

This paper We give an alternative to choosing between clutter and unsoundness. We propose an analysis that indicates which sorts are safe to erase, leaving ideally only a few sorts left that need to be translated using one of the first two methods.

The problem with sort erasure is that it forces all sorts to use the same domain. If the domains all had the same size to start with, there is no problem. But if the sorted formula only has models where some domains have different sizes than others, the sort erasure makes the formula unsatisfiable. We formulate this observation in the following lemma:

Lemma 1. *The following statements about a many-sorted first-order formula φ are equivalent:*

1. *There is an unsorted model with domain D for the sort-erased version of φ.*
2. *There is a model of φ where the size of each domain is $|D|$.*

Proof. (sketch) The interesting case relies on the observation that, if there is a sorted model in which all domains have the same size, then there also is a model in which all the domains are identical, from which it is trivial to construct an unsorted model. □

Our main contribution is a *monotonicity inference* calculus that identifies the so-called *monotone* sorts in a problem. If all sorts in a satisfiable sorted problem are monotone, then it is guaranteed that there will always be models for which all domains have the same size, in which case sort erasure is sound. The sorts that cannot be shown monotone will have to be made monotone first by introducing sort predicates or functions, but for these sorts only.

2 Monotonicity Calculus for First-Order Logic

Monotonox exploits *monotonicity* in the formula we are translating to produce a more efficient translation than the naive one. The purpose of this section is to explain what monotonicity is and how to infer it in a formula; section 3 explains how we use this information in Monotonox.

Before tackling monotonicity in a sorted setting, we first describe it in an unsorted one. We do this just because the notation gets in the way of the ideas when we have sorts.

2.1 Monotonicity in an Unsorted Setting

We start straight away with the definition of monotonicity. Monotonicity is a semantic property rather than a syntactic property of the formula.

Definition 1 (monotonicity, unsorted). *An unsorted formula φ is* monotone *if, for all $n \in \mathbb{N}$, whenever φ is satisfiable over domains of size n, it is also satisfiable over domains of size $n + 1$.*

An immediate consequence is that if a monotone formula is satisfiable over a finite domain, it is also satisfiable over all bigger finite domains.

Remark 1. Several common classes of formulae are monotone:

- Any unsatisfiable formula is monotone because it trivially satisfies our definition. The same goes for any formula that has no finite models.
- Any valid formula is monotone because it has a model no matter what the domain is.
- A formula that does not use $=$ is monotone, as we will see later.

What about a non-monotone formula? The simplest example is $\forall X, Y.\, X = Y$, which is satisfied if the domain contains a single element but not if it contains two. We will see later that equality is the single source of nonmonotonicity in formulae.

Monotonicity allows us to take a model of a formula and get from it a model over a bigger domain. Although it is not obvious from our definition, this is even the case if we want to get an infinite model.

Lemma 2 (monotonicity extends to infinite domains). *φ is monotone iff, for every pair of domains D and D' such that $|D| \leq |D'|$, if φ is satisfiable over D then φ is satisfiable over D'.*

Proof. (sketch) If φ is monotone and has a finite model then it has models of unbounded size; by compactness it has an infinite model. The lemma follows from this property and Löwenheim-Skolem.

□

Monotonicity is not decidable We can see from remark 1 that monotonicity is related to satisfiability, so we should not expect it to be decidable. Indeed it is not.[2] This does not mean we should give up on inferring monotonicity, just that we cannot *always* infer that a formula is monotone. The calculi we present later only answer "yes" if a formula is monotone but may answer "no" for a monotone formula too.

2.2 Monotonicity in a Many-Sorted Setting

Everything above generalises to sorted formulae, with the complication that we now have to talk about a formula being monotone in a particular sort. Informally, φ is monotone in the sort α if, given a model of φ, we can add elements to the domain of α while preserving satisfiability.

We use the notation $D(\alpha)$ for the domain of sort α. The formal definition mimics the one from the last section:

Definition 2 (monotonicity, sorted). *A sorted formula φ is* monotone in the sort α *if, whenever φ is satisfiable over D, and we are given D' such that*

[2] The proof works by encoding a given Turing machine by a formula that has a finite model of size k iff the Turing machine halts in exactly k steps. Thus if the Turing machine halts then the formula has a finite model at exactly one domain size and is therefore not monotone; if the Turing machine does not halt then the formula is finitely unsatisfiable and therefore monotone.

– $|D(\alpha)|$ *is finite, and* $|D(\alpha)| + 1 = |D'(\alpha)|$*, and*
– $D'(\beta) = D(\beta)$ *for all* $\beta \neq \alpha$*,*

then φ *is satisfiable over* D'*.*

Once again, we only consider taking a finite domain and adding a single element to it. The lemma from the last section still holds:

Lemma 3 (monotonicity extends to infinite domains (sorted)). φ *is monotone in* α *iff, whenever* φ *is satisfiable over* D*, and we are given* D' *such that*

– $|D'(\alpha)| \geq |D(\alpha)|$*, and*
– $D'(\beta) = D(\beta)$ *for all* $\beta \neq \alpha$*,*

then φ *is satisfiable over* D'*.*

The key insight of Monotonox is that *sort erasure is safe if the formula is monotone* in all sorts:

Theorem 1 (monotone formulae preserve satisfiability under erasure). *If* φ *is a many-sorted monotone formula, then* φ *and its sort-erasure are equi-satisfiable.*

Proof. By lemma 1, it is enough to show that from a model of φ we can construct a model where all domains are the same size. By lemma 3 we can do this by extending all the domains to match the size of the biggest domain. □

Remark 2. Notice that this construction preserves finite satisfiability, which is important when we are going to use a finite model finder on the problem.

Going back to our monkeys example, the formula is monotone in the sort *banana* (you can always add a banana to the model) but not in the sort *monkey* (if we have k monkeys and $2k$ bananas, we may not add another monkey without first adding two bananas). In section 3.2 we will see that this means we only need to introduce a sort predicate for the sort *monkey*.

2.3 A Simple Calculus for Monotonicity Inference

We now present two calculi for inferring monotonicity of a formula. In both calculi we assume that the formula is in CNF.

Our first calculus is based on the key observation that any formula that does not use equality is monotone. To see why, suppose we have a model over domain D of a formula φ, and we want to add a new element to D while preserving satisfiability. We can do this by taking an existing domain element $e \in D$ and making the new element e' behave identically to e, so that for all unary predicates P, $P(e)$ is true iff $P(e')$ is true, and for all unary functions f, $f(e) = f(e')$, and similarly for predicates and functions of higher arities. If the formula does not use equality, e and e' cannot be distinguished. Thus, the addition of a new domain element preserves satisfiability of the formula.

On the other hand, with equality present, the addition of a new element to the domain may make a previously satisfiable formula unsatisfiable. For example, $\forall X, Y.\, X = Y$ has a model with domain size 1, but it is not satisfiable for any larger domain size. We cannot make the new domain element behave the same as the old domain element because equality can distinguish them.

However, not all occurrences of equality have this problem. The following examples of equality literals are all monotone:

1. Negative equality (by increasing the size of the domain, more terms may become unequal but previously unequal terms will not become equal).
2. Equality where neither side is a variable (i.e. both sides are functions or constants, possibly with variable arguments). This is because, by using the strategy above for extending the domain with a new element, no function ever returns the new element, so the new element is never tested for equality.
3. Equality over a sort α is monotone in any sort β different to α. (The satisfiability of $t_1 = t_2$, where t_1 and t_2 have sort α is unaffected by the addition of new elements to the domain of β).

Thus, the only problematic literal for monotonicity in the sort α is positive equality over α where either side of the equality is a variable.

Safe terms We call a term *safe* in a sort α if, whenever we add a new element to the domain of α, the term never evaluates to this element. If the terms occurring on each side of an equality literal are both safe, the satisfiability of the literal is unaffected by the addition of new domain elements. Since positive equality literals are the only possible sources of nonmonotonicity, we can infer monotonicity of a formula by showing that all arguments of positive equality literals are safe. By the examples above, a term is safe in the sort α if it is not a variable, or it has a sort different to α. The simple calculus exploits these facts with the following rules:

1. $\varphi_1 \vee \varphi_2$ is monotone in α iff φ_1 and φ_2 are monotone in α.
2. $\varphi_1 \wedge \varphi_2$ is monotone in α iff φ_1 and φ_2 are monotone in α.
3. Any non-equality literal is monotone in any sort α.
4. $t_1 \neq t_2$ is monotone in any sort α.
5. $t_1 = t_2$ is monotone in α if t_1 and t_2 are safe in α, i.e., are not variables or are not of sort α.

Let us try out the simple calculus on the hungry monkeys in Example 1. The formula is monotone in *monkey* iff all of its clauses are monotone in *monkey*, and similarly for *banana*. Clauses (1) and (2) are monotone in both sorts, because the clauses do not contain equality. (3) is monotone in both sorts, because the clause does not contain positive equality. (4) is monotone in *banana*, because there is no equality between *banana* elements. The calculus does not let us infer monotonicity of *monkey* in this clause, because of the occurrence of an equality literal with two variables of sort *monkey*. Thus, the formula is monotone in *banana*, but not in *monkey*. This is consistent with our previous observation that we can add more *banana* elements without affecting satisfiability, but this is not the case for *monkey* elements.

2.4 Improved Calculus

There are many cases when our first calculus is not able to prove monotonicity. For example, suppose we change the problem so that some monkeys are not hungry and do not need bananas:

Example 2.

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \implies \mathsf{owns}(M, \mathsf{banana}_1(M))) \tag{5}$$

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \implies \mathsf{owns}(M, \mathsf{banana}_2(M))) \tag{6}$$

$$\forall M \in \mathit{monkey}.\, (\mathsf{hungry}(M) \implies \mathsf{banana}_1(M) \neq \mathsf{banana}_2(M)) \tag{7}$$

$$\begin{aligned} \forall M_1, M_2 \in \mathit{monkey}, B \in \mathit{banana}.& \\ ((\mathsf{hungry}(M_1) \wedge \mathsf{hungry}(M_2) \wedge \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B)& \\ \implies M_1 = M_2)& \end{aligned} \tag{8}$$

It is not hard to see that, given a model of the axioms, we can always add an extra monkey, by making that monkey not be hungry. Thus, the above formula is monotone in *monkey*. However, our simple calculus can not infer this, because of the use of positive equality between two variables of sort *monkey* in (8). In this section we remedy the problem by extending the calculus.

In the simple calculus, the strategy for extending a model while preserving finite satisfiability was to pick an existing element e in the domain, and let any new domain element "mimic" e. This strategy does not work for clause (8) in Example 2: if we happen to pick an e such that $\mathsf{hungry}(e)$ is true, then this strategy will add an extra hungry monkey to the domain, which does not preserve finite satisfiability. In our improved calculus we can make use of alternative strategies for extending the model, which allows us to infer monotonicity in cases such as this.

Extension rules In the improved calculus, we nominate some predicates to be *"true-extended"* and some to be *"false-extended"* in each sort α. If a predicate is neither true-extended nor false-extended, we say that it is *"copy-extended"*. When extending the model with a new domain element e', if a predicate P is true-extended, we make P return true whenever any of its arguments is e'; likewise if it is false-extended we make it return false if any of its arguments is e'. Copy-extended predicates behave as in the simple calculus.

Guard literals We say that a literal $P(\ldots)$ in a clause C *guards* an occurrence of a variable $X \in \alpha$ in C if X is one of the arguments of that literal and P is true-extended in α. Similarly, a literal $\neg P(\ldots)$ in C with X among its arguments guards occurrences of X in C if P is false-extended in α. We call the literal $P(\ldots)$ or $\neg P(\ldots)$ in this case a *guard literal*. The idea is that when X is instantiated with the new domain element, the guard literal is true, hence satisfiability of the clause is preserved. This allows us to infer that a clause involving positive equality between variables is monotone, if those variables are guarded. For example, in the clause (8) in Example 2, the two variables M_1 and

M_2 occurring in the equality literal are guarded by the predicate hungry, which we can make false for any new elements of sort *monkey* that we add.

Furthermore, $X \neq t$ guards X if t is not a variable: the clause $X \neq t \vee \varphi[X]$ is equivalent to $X \neq t \vee \varphi[t]$, in which X does not appear unsafely.[3]

Contradictory extensions When considering formulae, things get more problematic: if we add an axiom

$$\forall M \in monkey.\, \mathsf{hungry}(M) \tag{9}$$

to the formula in Example 2, we cannot add non-hungry monkeys to the domain, so the problem is no longer monotone in the sort *monkey*. For the clause (8) to be monotone, M_1 and M_2 must be guarded, which means that the predicate hungry must be false-extended. But extending hungry with false will not preserve satisfiability of the clause (9).

The new extension rules thus require some caution. If a predicate P is false-extended, then any occurrence of a variable X in the literal $P(\ldots)$ needs guarding just like it does in an equality literal $X = t$. Likewise, if P is true-extended, any occurrence of a variable X in the literal $\neg P(\ldots)$ needs guarding. This is illustrated in Example 3:

Example 3.

$$\forall X.\, (P(X) \Longrightarrow X = t) \tag{10}$$
$$\forall X.\, (Q(X) \Longrightarrow P(X)) \tag{11}$$

(10) requires P to be false-extended, because the occurrence of X in the positive equality literal needs guarding. But if $P(X)$ is false whenever X is instantiated with a new domain element, then Q must be false-extended in order to satisfy (11).

An occurrence of a variable X is problematic if it occurs in a literal of one of the following forms:

- $X = t$ or $t = X$
- $P(\ldots, X, \ldots)$ where P is false-extended
- $\neg P(\ldots, X, \ldots)$ where P is true-extended

In that case, we need to guard X for the formula to be monotone in X's sort.

The improved calculus infers monotonicity of a formula in α iff there is a consistent extension of predicates that guards all such variable occurrences.

2.5 Monotonicity Inference Rules of the Improved Calculus

Notation In the following, we shall use the abbreviation K to denote a function from predicates to the extension methods $\{\mathsf{true}, \mathsf{false}, \mathsf{copy}\}$. We call such a K a *context*. Furthermore, we use the notation $K \rhd_\alpha \varphi$ to mean that φ is monotone in the sort α, given the context K.

[3] This even holds if X is a subterm of t.

Formulae A formula φ is monotone with context K in the sort α iff all of its clauses are monotone with K in α:

$$\frac{K \rhd_\alpha C_1 \ \cdots \ K \rhd_\alpha C_n}{K \rhd_\alpha C_1 \wedge \ldots \wedge C_n}$$

Clauses In the rule for clauses, we must also consider the set Γ of variables that are guarded in the clause. We write $\Gamma, K \rhd_\alpha l$ if l is monotone with K in α, given that the variables in Γ are guarded. A clause is monotone with context K in the sort α if all of its literals are monotone with K in α, given Γ:

$$\frac{\Gamma = \bigcup_{i=1}^{n} \mathsf{guarded}(K, l_i) \qquad \Gamma, K \rhd_\alpha l_1 \ \cdots \ \Gamma, K \rhd_\alpha l_n}{K \rhd_\alpha l_1 \vee \ldots \vee l_n}$$

where $\mathsf{guarded}(K, l)$ is defined as

$\mathsf{guarded}(K, P(t_1 \ldots t_n)) = \{X \| X \in \{t_1 \ldots t_n\}, X \text{ is a variable}\}$ if $K(P) = \mathsf{true}$,
$\mathsf{guarded}(K, \neg P(t_1 \ldots t_n)) = \{X \| X \in \{t_1 \ldots t_n\}, X \text{ is a variable}\}$ if $K(P) = \mathsf{false}$,
$\mathsf{guarded}(K, X \neq t) = \{X\}$ if X is a variable and t is not,
$\mathsf{guarded}(K, l) = \emptyset$ otherwise.

Literals We have the following rules for monotonicity inference of literals:

$$\frac{}{\Gamma, K \rhd_\alpha t \neq_\beta u} \ \textbf{(1)} \qquad \frac{\beta \neq \alpha}{\Gamma, K \rhd_\alpha t =_\beta u} \ \textbf{(2)} \qquad \frac{\mathsf{safe}(\Gamma, t, \alpha) \ \ \mathsf{safe}(\Gamma, u, \alpha)}{\Gamma, K \rhd_\alpha t =_\alpha u} \ \textbf{(3)}$$

$$\mathsf{safe}(\Gamma, t, \alpha) = \begin{cases} t \in \Gamma & \text{if } t \text{ is a variable of sort } \alpha, \\ \mathsf{true} & \text{otherwise.} \end{cases}$$

(1) Negative equality is always monotone. (2) Equality in a sort β is monotone in any sort α that is different to β. (3) Equality between two terms is monotone if the terms are non-variables, or are guarded in the clause.

$$\frac{\mathsf{safe}(\Gamma, t_1, \alpha) \ \cdots \ \mathsf{safe}(\Gamma, t_n, \alpha)}{\Gamma, K \rhd_\alpha P(t_1, \ldots, t_n)} \ \textbf{(4)} \qquad \frac{\mathsf{safe}(\Gamma, t_1, \alpha) \ \cdots \ \mathsf{safe}(\Gamma, t_n, \alpha)}{\Gamma, K \rhd_\alpha \neg P(t_1, \ldots, t_n)} \ \textbf{(5)}$$

(4,5) A predicate literal is monotone in α if all of its variable arguments of sort α are guarded in the clause in which the literal occurs.

$$\frac{K(P) \in \{\mathsf{true}, \mathsf{copy}\}}{\Gamma, K \rhd_\alpha P(t_1, \ldots, t_n)} \ \textbf{(6)} \qquad \frac{K(P) \in \{\mathsf{false}, \mathsf{copy}\}}{\Gamma, K \rhd_\alpha \neg P(t_1, \ldots, t_n)} \ \textbf{(7)}$$

(6) A positive occurrence of a predicate is monotone if the predicate is true-extended or copy-extended. (7) A negative occurrence of a predicate is monotone if the predicate is false-extended or copy-extended.

It is not immediately clear how to implement the above rules, since there is no obvious way to infer the context K. We see in section 3.1 that we can do this using a SAT-solver.

2.6 NP-completeness of the Improved Calculus

The improved calculus allows us to infer monotonicity in more cases. However, inferring monotonicity with it is NP-complete. We show NP-hardness by reducing CNF-SAT to a problem of inferring monotonicity in the calculus.

Given any propositional formula φ_{SAT} in CNF, we construct a formula φ_{MON} such that φ_{SAT} is satisfiable iff φ_{MON} is monotone. The idea is that a context that makes φ_{SAT} monotone corresponds to a satisfying assignment for φ_{SAT}.

For each positive literal l in φ_{SAT}, we introduce a unary predicate P_l in φ_{MON}. For negative literals $\neg l$, we define $P_{\neg l}(X)$ as $\neg P_l(X)$. We equip φ_{MON} with a single constant c. We translate each clause $(l_1 \vee ... \vee l_n)$ of φ_{SAT} into the following clause in φ_{MON}:

$$\forall X.\, P_{l_1}(X) \vee ... \vee P_{l_n}(X) \vee X = \mathsf{c}$$

Our calculus proves this clause monotone exactly when our context extends at least one of $P_{l_1}, .., P_{l_n}$ by true. Thus if we find a context that makes φ_{MON} monotone we may extract a satisfying assignment for φ_{SAT} by doing the following for each positive literal l of φ_{SAT}:

- If P_l is extended by true then let l be true.
- If P_l is extended by false then let l be false.
- If P_l is extended by copy then choose an arbitrary value for l.

The same method takes us from a satisfying assignment of φ_{SAT} to a context that makes φ_{MON} monotone.

3 Monotonox: Sorted to Unsorted Logic and Back Again

We have implemented the monotonicity calculus as part of our tool Monotonox. This section first shows how the calculus is implemented and then how monotonicity is exploited in translating between sorted and unsorted first-order logic.

3.1 Monotonicity Inference with Monotonox

We show in this section how to use a SAT-solver to implement the monotonicity calculus. The use of a SAT-solver is a reasonable choice, as we have seen previously that monotonicity inference in our calculus is NP-hard.

We encode the problem of inferring monotonicity of a formula φ as a SAT-problem, where a satisfying assignment corresponds to a context in our calculus.

We construct for each predicate P in φ two literals, p_T and p_F. The idea is that if p_T is assigned true, then P should be true-extended. If p_F is assigned true, then P should be false-extended. If both p_T and p_F are assigned false, then P should be copy-extended. Our task is to construct a propositional formula with these literals, that is satisfiable iff φ is monotone according to our calculus.

Formulae The SAT-encoding of a formula φ is the conjunction of SAT-encodings of the clauses of φ and the constraint that each predicate may not be extended by both true and false:

$$\mathsf{monotone}((C_1 \wedge .. \wedge C_n), \alpha) = \bigwedge_{i=1}^{n} \mathsf{monotone}(C_i, \alpha) \wedge \bigwedge_{P_i \in \varphi} \neg p_{i_F} \vee \neg p_{i_T}$$

Clauses The SAT-encoding of a clause C is the conjunction of SAT-encodings of the literals of C.

$$\mathsf{monotone}((l_1 \vee .. \vee l_n), \alpha) = \bigwedge_{i=1}^{n} \mathsf{monotone}((l_1 \vee .. \vee l_n), l_i, \alpha)$$

Literals The SAT-encoding of a literal may depend on the clause in which it occurs. In a positive equality literal, both of the terms must be safe. A negative equality literal is trivially monotone. An occurrence of a predicate is monotone if the predicate is extended in an appropriate way or its arguments are safe.

$$\mathsf{monotone}(C, l, \alpha) = \begin{cases} \mathsf{safe}(C, t_1, \alpha) \wedge \mathsf{safe}(C, t_2, \alpha) & \text{if } l \text{ is } t_1 = t_2, \\ \mathsf{true} & \text{if } l \text{ is } t_1 \neq t_2, \\ \neg p_F \vee \bigwedge_{i=1}^{n} \mathsf{safe}(C, t_i, \alpha) & \text{if } l \text{ is } P(t_1, \ldots, t_n), \\ \neg p_T \vee \bigwedge_{i=1}^{n} \mathsf{safe}(C, t_i, \alpha) & \text{if } l \text{ is } \neg P(t_1, \ldots, t_n), \end{cases}$$

A term t is safe in a clause if it is not a variable of the sort considered for monotonicity, or it is guarded by any of the literals in the clause.

$$\mathsf{safe}((l_1 \vee .. \vee l_n), t, \alpha) = \begin{cases} \bigvee_{i=1}^{n} \mathsf{guards}(l_i, t) & \text{if t is a variable of sort } \alpha \\ \mathsf{true} & \text{otherwise} \end{cases}$$

A literal l guards a variable X according to the rules that we discussed in section 2.4.

$$\mathsf{guards}(l, X) = \begin{cases} p_T & \text{if } l \text{ is of the form } P(\ldots, X, \ldots), \\ p_F & \text{if } l \text{ is of the form } \neg P(\ldots, X, \ldots), \\ \mathsf{true} & \text{if } l \text{ is of the form } X \neq f(\ldots) \text{ or } f(\ldots) \neq X, \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

If there is a satisfying assignment of the SAT-formula $\mathsf{monotone}(\varphi, \alpha)$, then there is a consistent extension of the predicates of φ (a context) that makes φ monotone in α, and vice versa. Monotonox uses MiniSat [6] to find out whether a satisfying assignment exists for each sort.

3.2 Translating Sorted to Unsorted Logic

To translate from a sorted problem to an unsorted problem, we use the principle that *monotone sorts can simply be erased*, but non-monotone sorts need to be encoded using, for example, a sort predicate. Thus our algorithm is as follows:

1. Analyse the formula to discover which sorts are monotone.
2. For each non-monotone sort, transform the formula by introducing a sort predicate or a sort function (according to the user's choice)—but do not erase the sort yet.
3. Erase all the sorts at once.

It makes no difference *which* method we use to encode the non-monotone sorts—predicates, functions or something else. We can in principle use sort predicates for some sorts and sort functions for others.

We justify the algorithm as follows: by adding sort predicates or functions for all the non-monotone sorts, we have transformed the input formula into an equisatisfiable formula *which is also monotone*.[4] Once we have this monotone formula then erasing all the sorts preserves satisfiability (theorem 1).

An example Suppose we take the first axiom of our running example, $\forall M \in monkey.\, \mathsf{owns}(M, \mathsf{banana}_1(M))$. As discussed, we know that the sort *banana* is monotone but the sort *monkey* is not. Thus we need to introduce a sort predicate or function for only the sort *monkey*. If we introduce a sort function—while still keeping the formula sorted—the new formula we obtain is $\forall M \in monkey.\, \mathsf{owns}(\mathsf{f}_{monkey}(M), \mathsf{banana}_1(\mathsf{f}_{monkey}(M))$.

Having done this, it is enough to erase the sorts from the formula (step 3 of the algorithm) and we obtain an unsorted formula which is equisatisfiable over each domain size to the original sorted formula, namely:

$$\forall M.\, \mathsf{owns}(\mathsf{f}_{monkey}(M), \mathsf{banana}_1(\mathsf{f}_{monkey}(M))$$

3.3 Translating Unsorted to Sorted Logic

The translation from unsorted to sorted formulae makes use of the same machinery, only in the reverse direction: given an unsorted problem ϕ, if we find a well-sorted problem ψ such that (1) erasing the sorts in ψ gives us back ϕ, and (2) all sorts in ψ are monotone, then (theorem 1) ϕ and ψ are equisatisfiable.

The problem is finding the sorted problem ψ. We can use an existing algorithm [4], that we call *sort unerasure* here, for this. Sort unerasure computes the *maximal typing* of an unsorted problem. It starts by creating unique sorts for all variable occurrences in the problem, and for all argument positions of predicate and function symbols, and for all results of function symbols. Then, it computes equivalence classes of sorts that should be equal to each other in order for the problem to be well-sorted, in the following way. Everywhere in the problem, whenever we apply a function symbol or predicate symbol P to a term t, we force the sort of the corresponding argument position of P to be in the same equivalence class as the result sort of t. Using a union/find algorithm, we get an algorithm that is close to linear time in complexity.

To sum up, the translation goes in three steps:

[4] In the case of sort predicates, our second calculus can infer monotonicity by false-extending the sort predicate; in the case of sort functions, our first calculus also can because no variable appears directly as the argument of an equality literal.

1. Compute candidate sorts for all symbols occurring in the problem (using sort unerasure), and create the corresponding sorted problem.
2. Use Monotonox to find out if all sorts in the resulting problem are monotone. If they are, we are done.
3. If there exists any sort that cannot be shown monotone, then give up. We simply return the unsorted problem as a sorted problem with one sort.

In practice, there is more we can do in step 3 than giving up. One has to constrain the sorted formula so that (1) the domains of all non-monotone sorts have the same size, and (2) no monotone sort's domain can be bigger than a non-monotone sort's domain. A finite model finder can easily implement these constraints; when theorem-proving, one can enforce size constraints between sorts by adding to the problem an injective function from the smaller sort to the bigger sort.

4 Results

The TPTP library [11] has recently been extended with many-sorted (so-called TFF) problems. Unfortunately, only 26 of these problems have more than one sort.[5] They break down as follows: 11 have no non-ground positive equality, which means that they are trivially monotone. Monotonox proves a further 5 monotone. 4 are monotone only because they have no finite models, a situation which we cannot detect but plan to in the future. 6 are truly not monotone.

Translating from unsorted to many-sorted logic, we applied sort unerasure to all 13610 unsorted TPTP problems,[6] finding 6380 problems with more than one sort, to which we applied our monotonicity inference. The results are as follows.

	Total problems	Total sorts	Monotone sorts	Other sorts[7]	Affected problems[8]	Monotone problems[9]
CNF problems						
Simple calculus	2598	19908	12317	7591	2446	592
Improved calculus			12545	7363	2521	726
Full first-order problems						
Simple calculus	3782	91843	85025	6818	3154	1034
Improved calculus			88645	3198	3715	1532

Running times None of the tests above took more than a few seconds. Monotonicity inference was not more expensive than the sort unerasure algorithm.

[5] TFF adds both sorts and arithmetic to TPTP; the vast majority of the problems so far only test arithmetic, so only have one sort.
[6] Excluding the so-called `SYN` problems that just test syntax.
[7] Sorts that we couldn't infer monotone (including sorts that are truly not monotone).
[8] Problems where at least one sort was inferred monotone.
[9] Problems where all sorts were inferred monotone.

5 Conclusions and Future Work

We have introduced the concept of *monotonicity*, and applied it to the problem of translating between many-sorted and unsorted first-order logic. Detecting monotonicity of a sort is not decidable, but we have introduced two algorithms approximating the answer, one linear in the size of the problem, and one improved algorithm solving an NP-complete problem using a SAT-solver. Our results show that the improved algorithm detects many cases of monotonicity, and that the NP-completeness is not a problem in practice.

For future work, we plan to integrate our previous work on finite unsatisfiability detection [3] with monotonicity detection—any sort which must have an infinite domain is monotone. We expect this method to improve monotonicity detection for typical problems that have been translated from higher-order logics with recursive datatypes, such as lists. Moreover, we are working on generalising guards to arbitrary literals.

Finally, we plan to use the translation from unsorted to many-sorted logic to populate the typed section of the TPTP benchmark library.

References

1. Abel, A., Coquand, T., Norell, U.: Connecting a logical framework to a first-order logic prover. In: Gramlich [7], pp. 285–301
2. Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. In: Giesl, J., Hähnle, R. (eds.) IJCAR. Lecture Notes in Computer Science, vol. 6173, pp. 91–106. Springer (2010)
3. Claessen, K., Lilliestr̈om, A.: Automated inference of finite unsatisfiability. Journal of Automated Reasoning (2011), http://dx.doi.org/10.1007/s10817-010-9216-8, 10.1007/s10817-010-9216-8
4. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Proc. of Workshop on Model Computation (MODEL) (2003)
5. Enderton, H.B.: A Mathematical Introduction to Logic, chap. 4.3 (Many-Sorted Logic). Academic Press, second edn. (January 2001)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
7. Gramlich, B. (ed.): Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3717. Springer (2005)
8. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
9. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. Journal of Universal Computer Science 5(3), 73–87 (1999)
10. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich [7], pp. 48–64
11. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
12. Wick, C.A., McCune, W.: Automated reasoning about elementary point-set topology. Journal of Automated Reasoning 5(2), 239–255 (1989)